下载APP

14 | 用户信息查询：如何解决网络软中断瓶颈问题？

2021-04-21 高楼

高楼的性能工程实战课 进入课程 >

讲述：高楼
时长 19:14 大小 17.63M

你好，我是高楼。

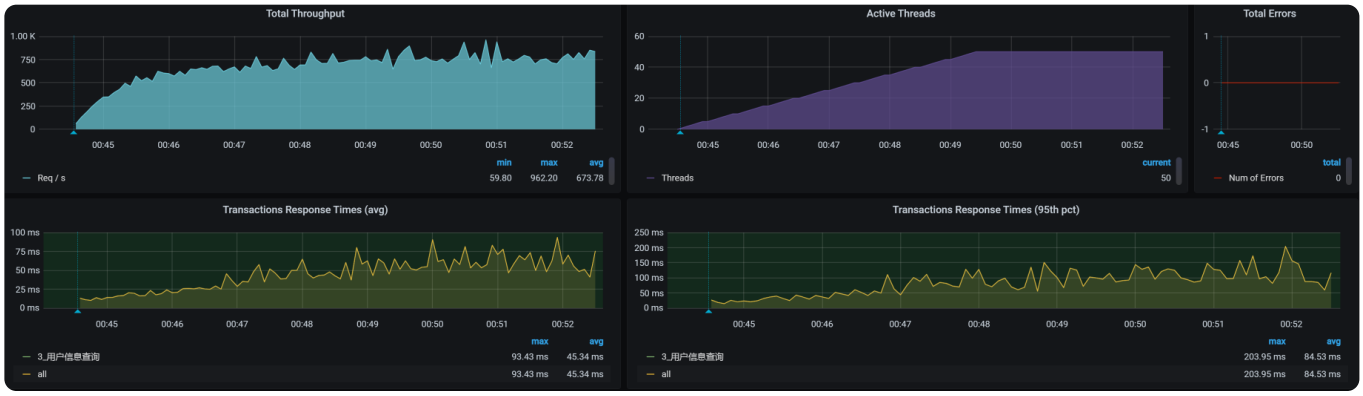
这节课我们接着来整另一个接口：用户信息查询。通过这个接口，我们一起来看看，当网络软中断过高时，会对 TPS 产生什么样的影响。其实对于这一点的判断，在很多性能项目中都会出现，而其中的难点就在于，很多人都无法将软中断跟响应时间慢和 TPS 所受到的影响关联起来。今天我就带你来解决这个问题。

同时，我也会给你讲解如何根据硬件配置及软件部署情况，做纯网络层的基准验证，以确定我们判断方向的正确性，进而提出具有针对性的优化方案。而我们最终优化的效果，通过 TPS 对比来体现。

压力数据

我们先来看用户信息查询的压力数据情况如何。因为我们现在测试的是单接口，而用户信息查询又需要有登录态，所以我们要先跑一部分 Token 数据出来，再执行用户信息查询接口。

准备好 Token 数据后，第一次用户信息查询如下：



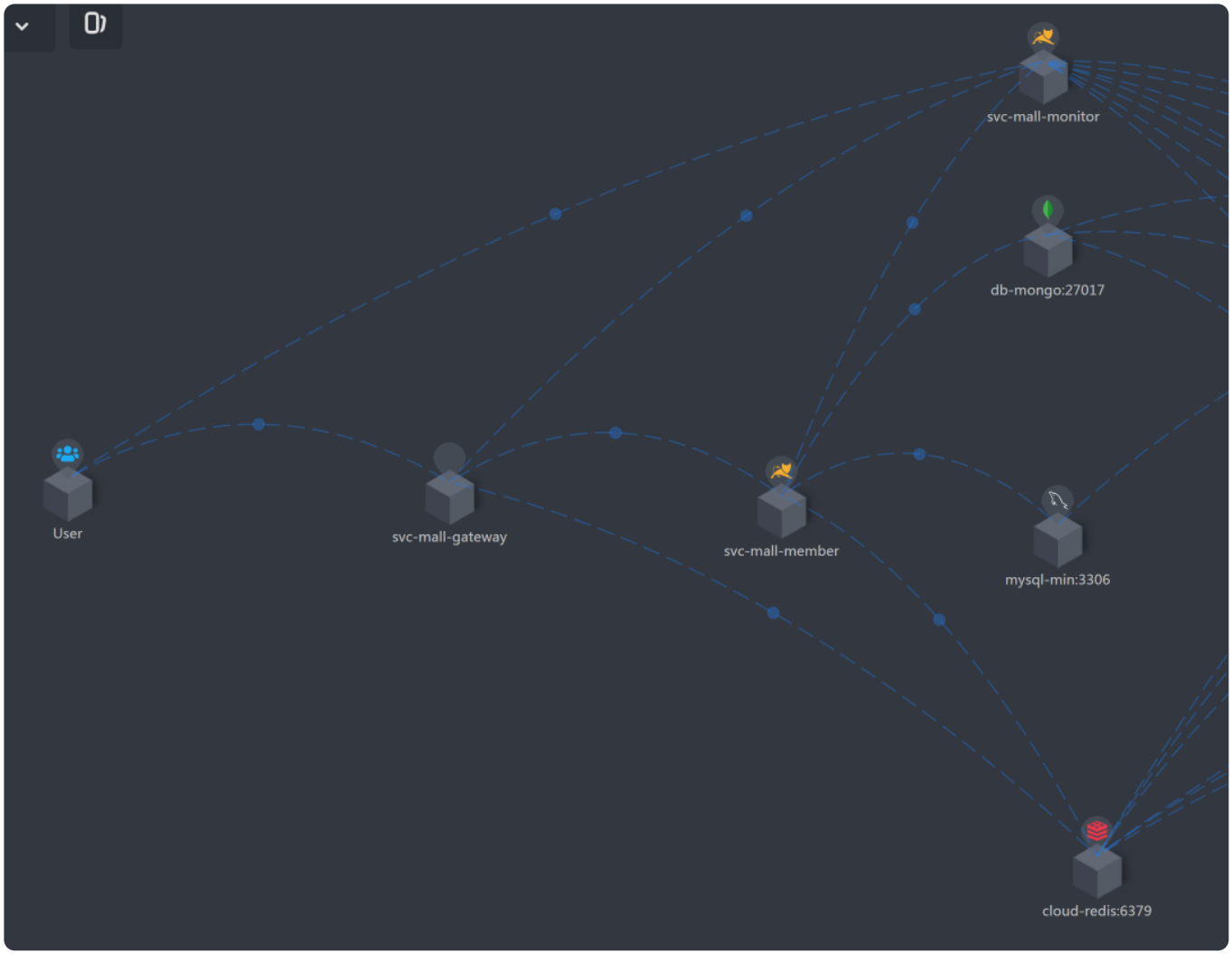
这个步骤只是试验一下，持续时间长是为了查找问题。从上图来看，这个接口的起点不错，已经达到 750 左右。

不过，性能瓶颈也比较明显：响应时间随着压力线程的增加而增加了，TPS 也达到了上限。对于这样的接口，我们可以调优，也可以不调优，因为这个接口当前的 TPS 可以达到我们的要求。只不过，本着“**活着不就是为了折腾**”的原则，我们还是要分析一下这个接口的瓶颈到底在哪里。

还是按照我们之前讲过的分析思路，下面我们来分析这个问题。

看架构图

从链路监控工具中，我们拉出架构图来，这样简单直接，又不用再画图了，真的是懒人必知技能。



从上图可以知道，用户信息查询的路径是 User - Gateway - Member - MySQL。

你也许会问，图中不是还有 Redis、MongoDB、Monitor 吗？是的，这些我们也要记在脑子里。这个接口用到了 Redis，如果接口有问题，变慢了，我们也要分析；MongoDB 并没有用上，所以我们不管它；Monitor 服务是 Spring Boot Admin 服务，我们也暂且不管它，后面需要用到的时候再说。

注意，这一步是分析的铺垫，是为了让我们后面分析时不会混乱。

拆分响应时间

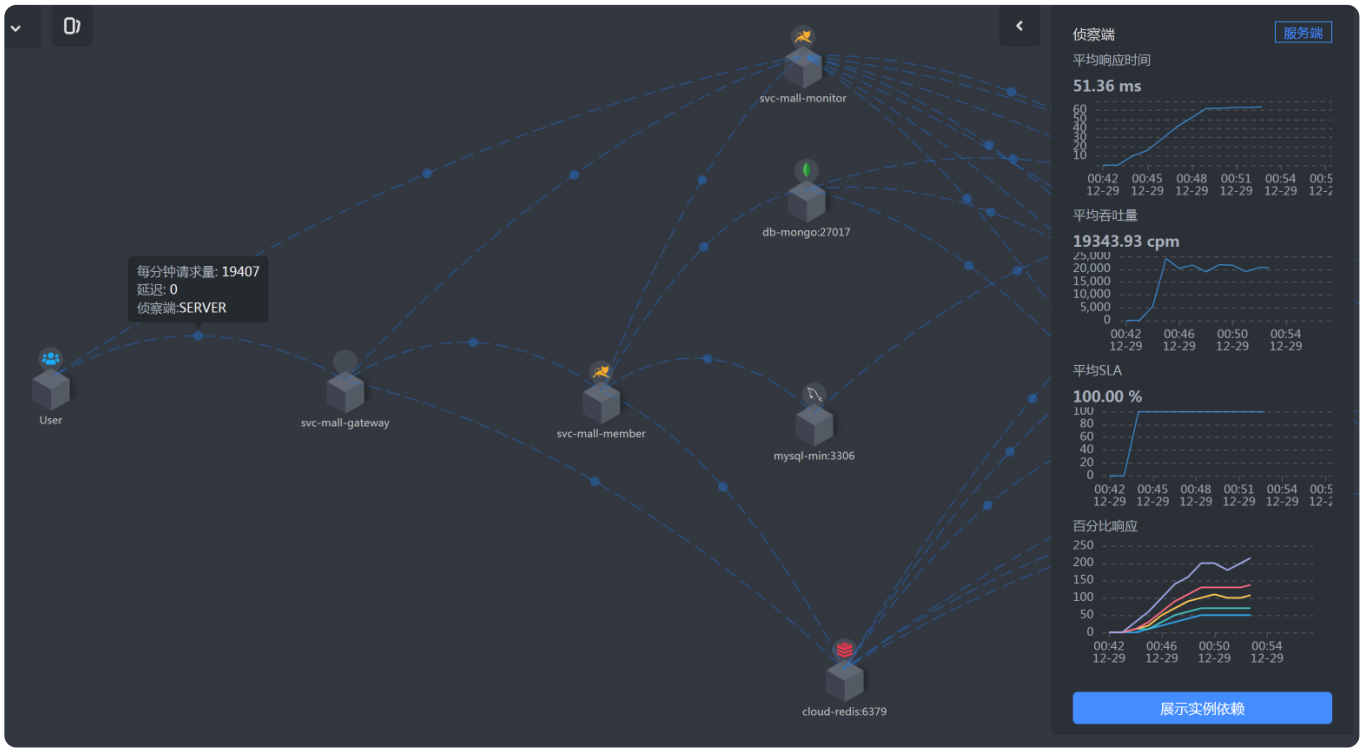
在场景数据中，我们明显看到响应时间慢了，那我们就要知道慢在了哪里。我们根据上面的架构图知道了用户信息查询接口的路径，现在就要拆分这个响应时间，看一看每一段消耗了多长时间。

如果你有 APM 工具，那可以直接用它查看每一段消耗的时间。如果你没有，也没关系，只要能把架构图画出来，并把时间拆分了就行，不管你用什么招。

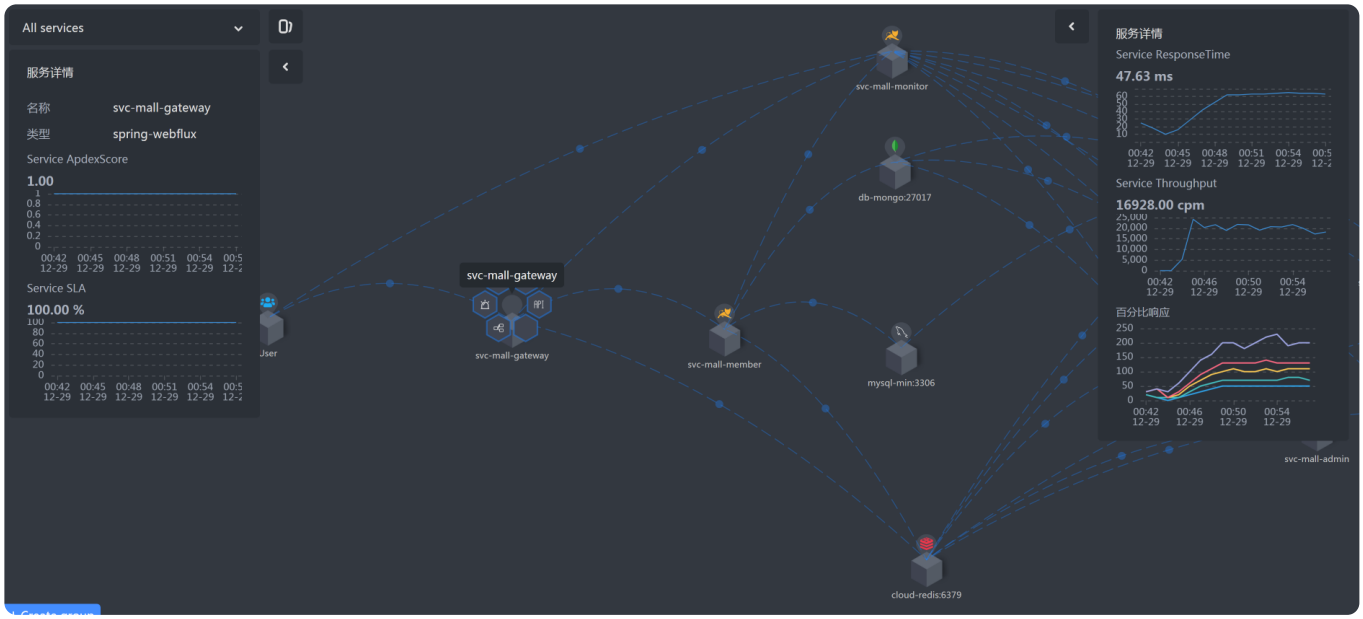
另外，我啰嗦一句，请你不要过分相信 APM 工具厂商的广告，咱们还是要看疗效。在追逐技术的同时，我们也需要理智地判断到底是不是需要。

具体的拆分时间如下：

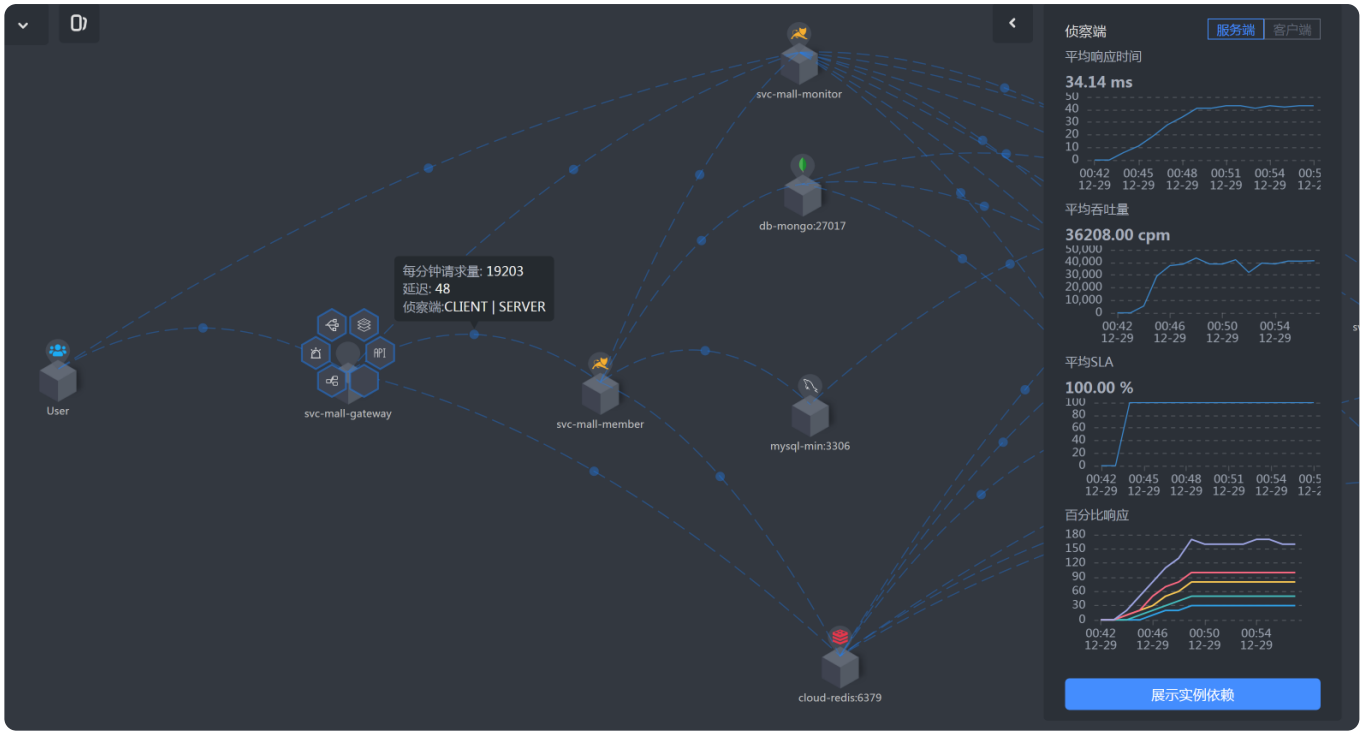
User -Gateway

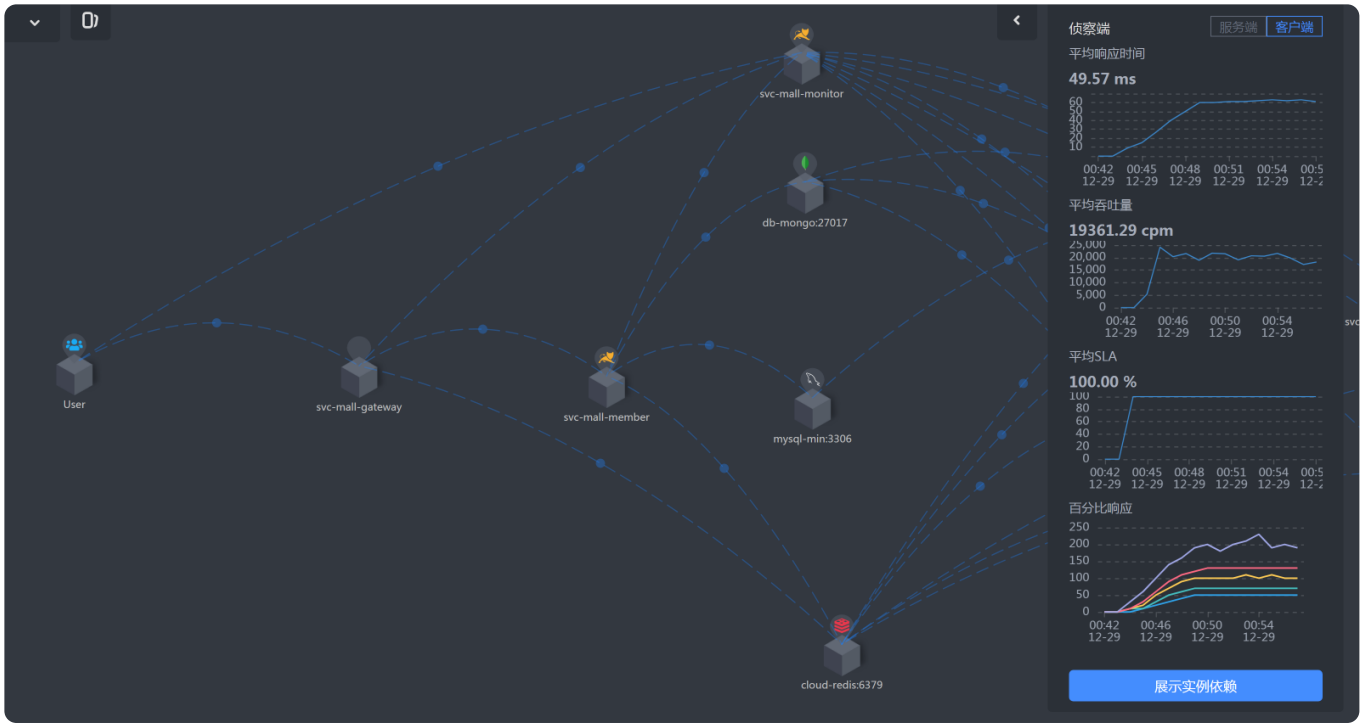


Gateway 上消耗的时间

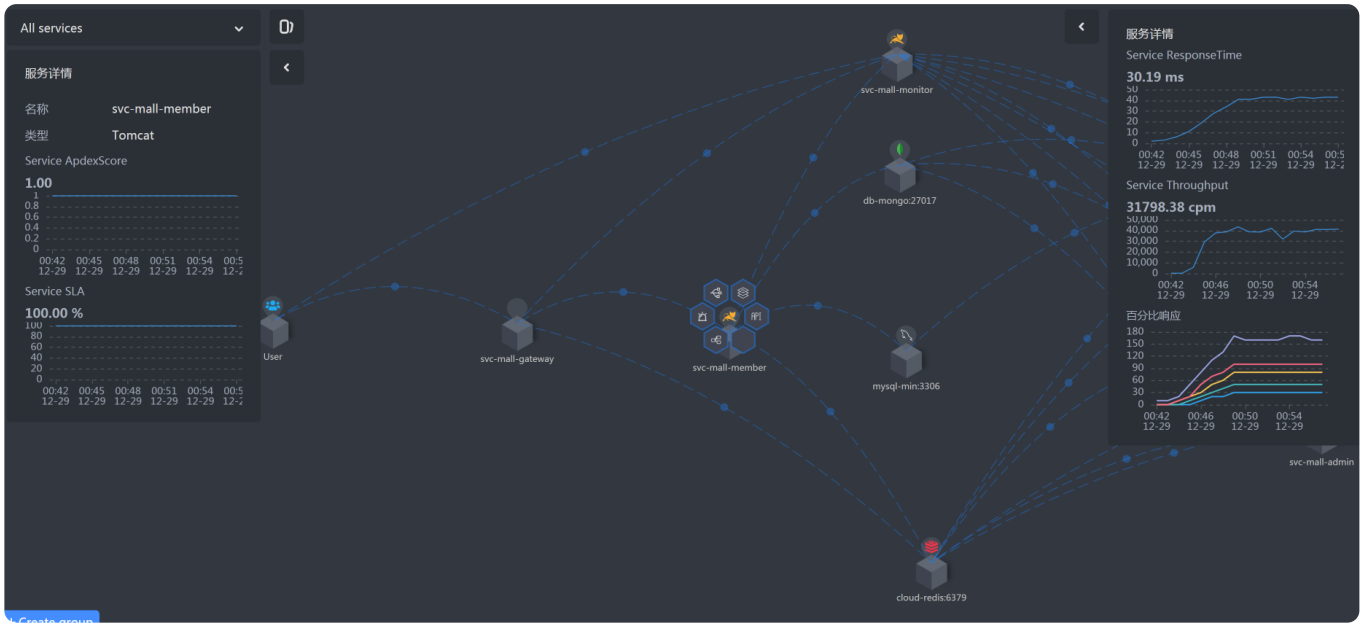


Gateway -Member

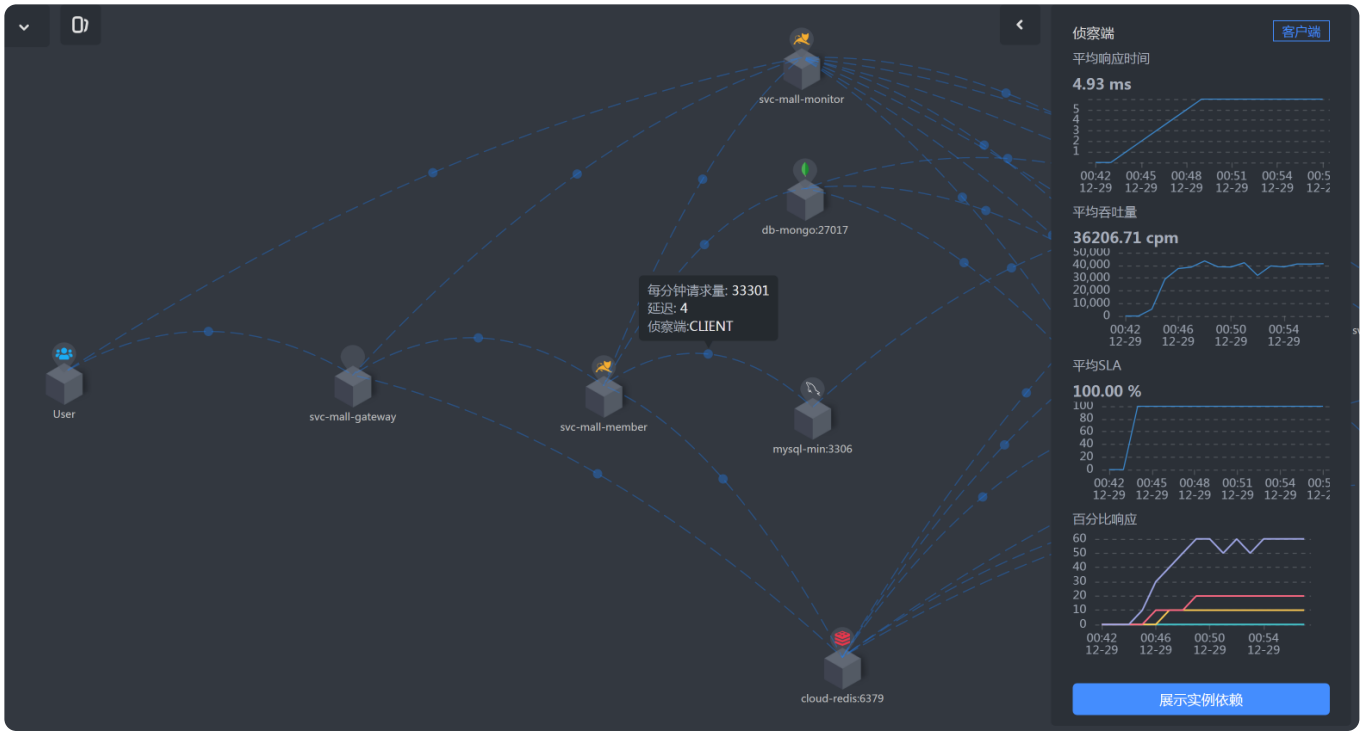




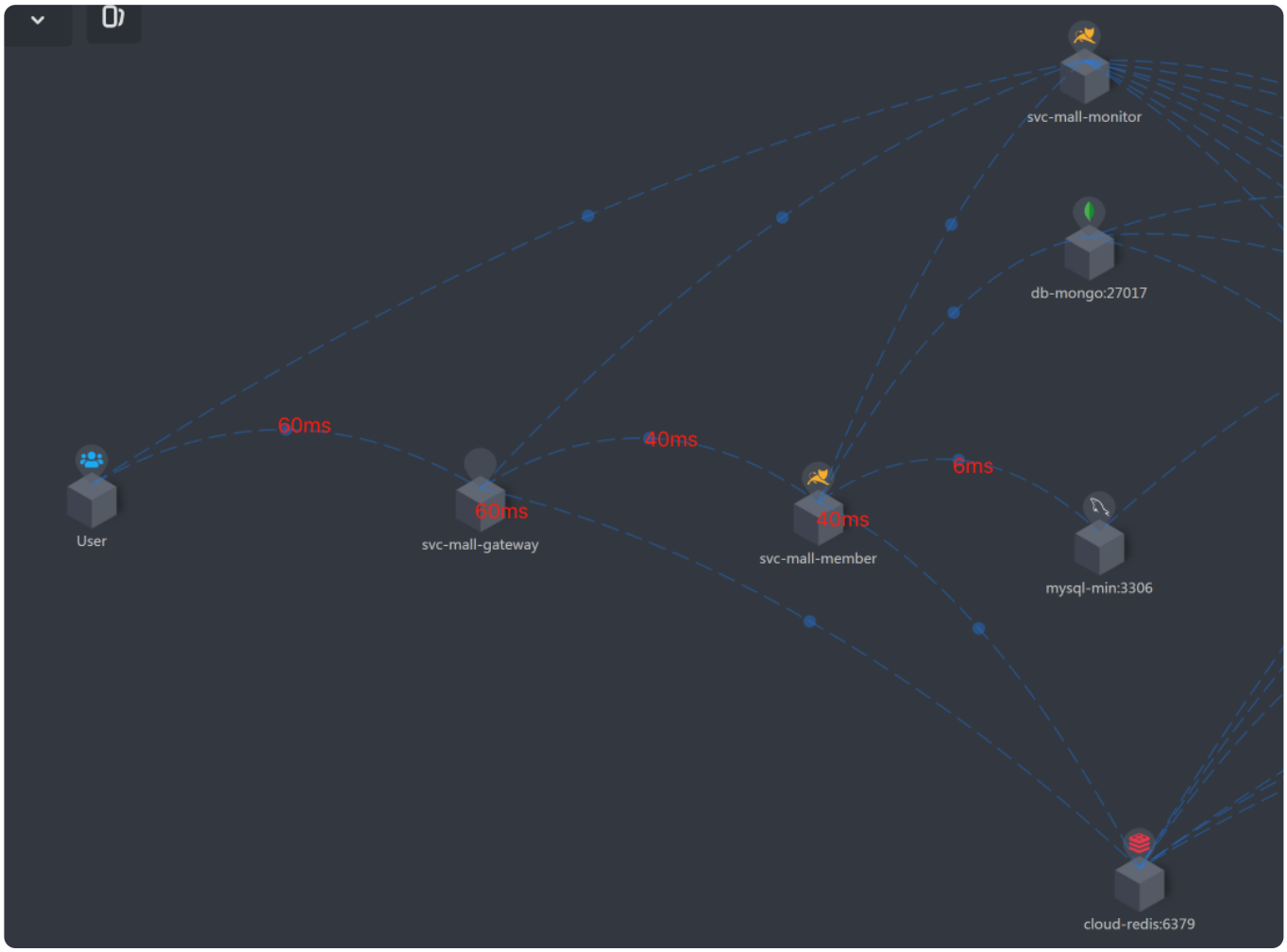
Member 上消耗的时间



Member 到 DB 的时间



我把上述拆分后的时间都整理到我们的架构图中：



看到这张图，思路变得特别清晰了，有没有？根据上图的时间拆分，我们明显看到 Member 服务上消耗时间更多一点，所以下一步我们去关注 Member 服务。

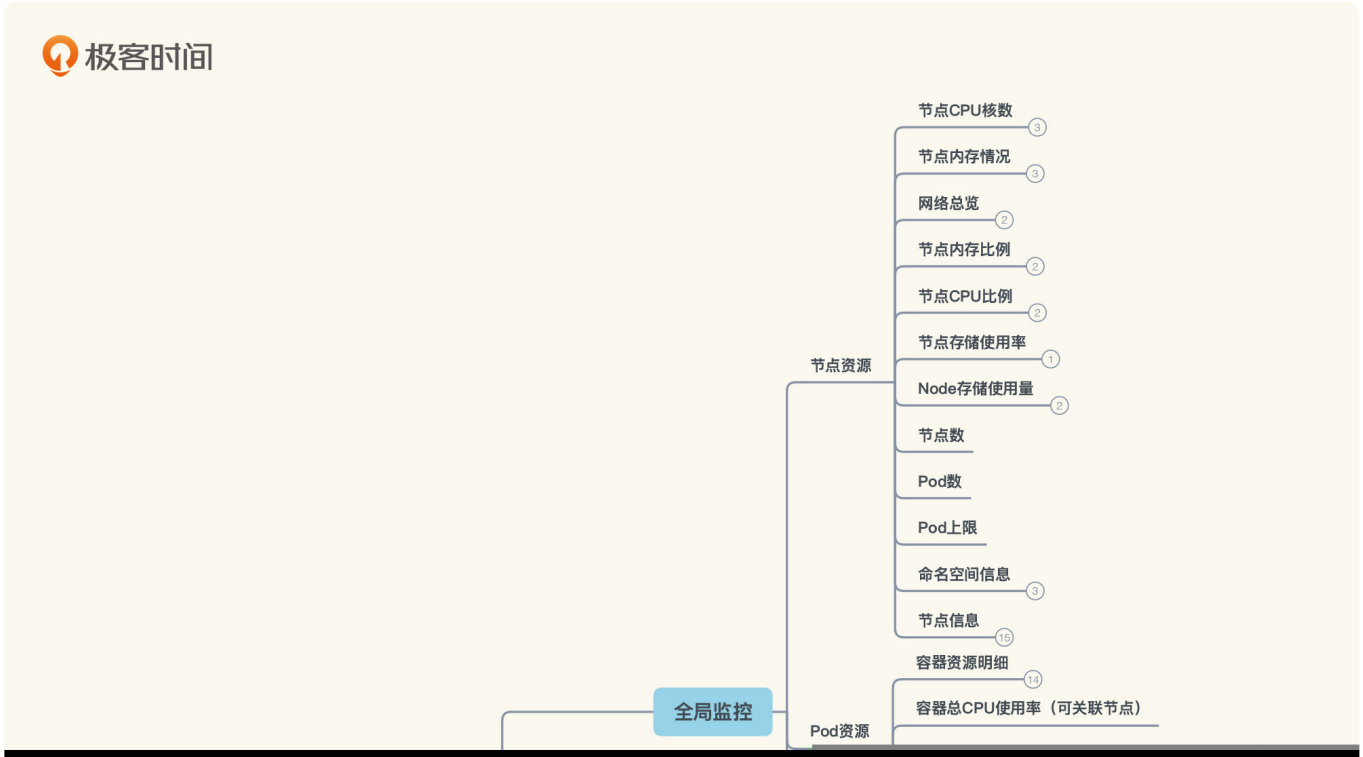
全局监控分析

还是一样，我们先看全局监控：

IP（链接到明细）	主机名	运行时间	内存	CPU核	5m负载	CPU使用率	内存使用率	分区使用率*	磁盘读取*	磁盘写入*	连接数	TCP_tw	下载带宽*	上传带宽*
k8s-master-1	k8s-master-1	1.31 week	7.64 GiB	4	0.40	10.23%	24.76%	29.95%	0 B/s	76.31 kB/s	309	21	1.15 Mbps	1.15 Mbps
k8s-master-2	k8s-master-2	1.31 week	7.64 GiB	4	0.86	16.43%	21.48%	29.95%	0 B/s	72.64 kB/s	271	25	246.15 kbps	269.35 kbps
k8s-master-3	k8s-master-3	1.31 week	7.64 GiB	4	0.28	7.12%	22.27%	29.95%	0 B/s	75.41 kB/s	280	20	309.96 kbps	425.51 kbps
k8s-worker-1	k8s-worker-1	1.55 day	15.51 GiB	6	3.04	26.81%	53.96%	32.55%	32.22 kB/s	191.05 kB/s	29	27 K	25.72 Mbps	39.47 Mbps
k8s-worker-2	k8s-worker-2	1.54 day	15.51 GiB	6	1.62	36.90%	47.83%	29.95%	9.21 kB/s	5.26 kB/s	29	27 K	20.51 Mbps	28.67 Mbps
k8s-worker-3	k8s-worker-3	1.55 day	15.51 GiB	6	0.13	4.52%	29.56%	30.89%	18.41 kB/s	6.30 kB/s	29	68	63.63 kbps	81.24 kbps
k8s-worker-4	k8s-worker-4	1.55 day	15.51 GiB	6	0.64	5.77%	19.64%	29.95%	9.21 kB/s	10.61 kB/s	47	145	915.16 kbps	1.11 Mbps
k8s-worker-5	k8s-worker-5	1.53 day	15.51 GiB	6	1.02	17.43%	37.34%	33.68%	23.02 kB/s	4.42 kB/s	29	64	5.42 Mbps	8.86 Mbps
k8s-worker-6	k8s-worker-6	1.53 day	15.51 GiB	6	0.65	10.83%	38.22%	35.48%	9.21 kB/s	2.51 kB/s	28	18	13.60 Mbps	14.31 Mbps
k8s-worker-7	k8s-worker-7	1.51 day	15.51 GiB	4	3.96	50.88%	37.17%	36.60%	23.01 kB/s	12.66 kB/s	28	163	13.33 Mbps	14.46 Mbps
k8s-worker-8	k8s-worker-8	1.51 day	15.51 GiB	4	26.65	93.40%	44.71%	41.94%	0 B/s	6.40 kB/s	32	688	26.34 Mbps	21.64 Mbps
k8s-worker-9	k8s-worker-9	1.51 day	15.51 GiB	6	1.84	25.57%	43.85%	29.95%	27.62 kB/s	10.46 kB/s	28	23	10.08 Mbps	16.66 Mbps

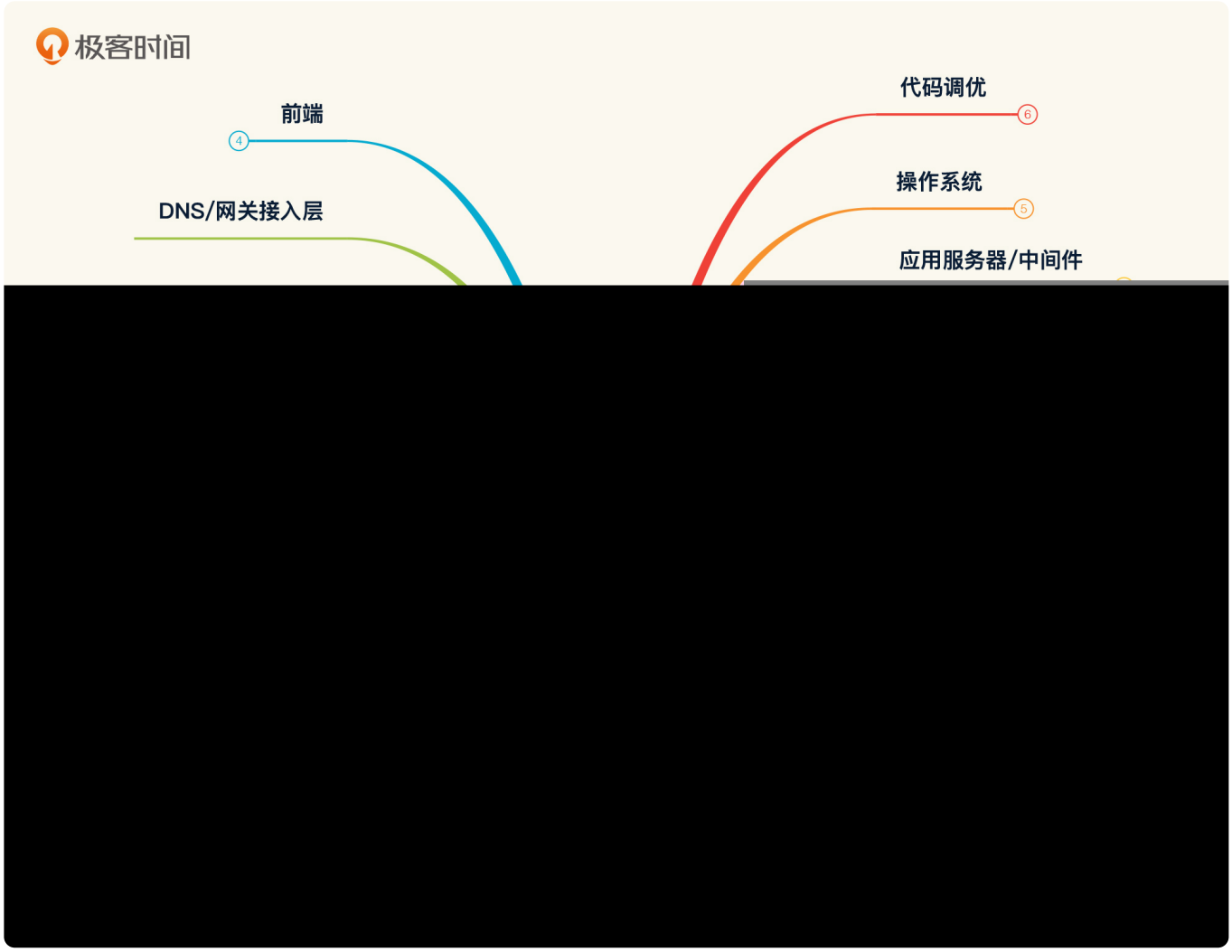
其中，worker-8 的 CPU 用得最多，我们先从这里下手。

这里我要跟你强调一下，**在全局监控的思路中，不是说我们看了哪些数据，而是我们要去看哪些数据**。这时候，你就必须先有一个全局计数器的东西。比如说在 Kubernetes 里，我们就要有这样的思路：



也就是说，**我们要先把全局监控的计数器都罗列出来，然后再一个一个查去。**

其实，这里面不止是罗列那么简单，它还要有相应的逻辑。那怎么弄懂这个逻辑呢？这就要依赖于性能分析人员的基础知识了。我经常说，要想做全面的性能分析，就必须具备计算机基础知识，而这个知识的范围是很大的。之前我画过一张图，现在我做了一些修正，如下所示：



图中这些内容，都是我们做性能分析时会遇到的东西。有人可能会说，这些已经远远超出性能工程师的技能范围了。所以我要再强调一下，我讲的一直都是性能工程。在整个项目的性能分析中，我并不限定技术的范围，只要是用得上，我们都需要拿出来分析。

前面我们说是 worker-8 上的 CPU 资源用得最多，所以我们来查一下被测的服务，也就是 Member 服务，是不是在 worker-8 上。

```
[root@k8s-master-2 ~]# kubectl get pods -o wide
NAME                                    READY   STATUS    RESTARTS   AGE    IP               NODE           NOMINATED NODE   READINESS GATES
cloud-nacos-registry-5d599dc8f8-t6js4   1/1     Running   0          36h    10.100.231.241   k8s-worker-8   <none>           <none>
cloud-rabbitmq-bb7686dfc-h8c22          1/1     Running   5          30d    10.100.79.188    k8s-worker-4   <none>           <none>
cloud-redis-5d7f7f79d6-wbfhx            2/2     Running   0          65m    10.100.231.220   k8s-worker-8   <none>           <none>
cloud-sentinel-dashboard-846bb4c76c-vqkxf   1/1     Running   2          30d    10.100.79.183    k8s-worker-4   <none>           <none>
db-mongo-79497bfc5b-fcx98               1/1     Running   0          36h    10.100.18.217    k8s-worker-5   <none>           <none>
```

从上图看，Member 服务确实是在 worker-8 上。

那下一步我们就要进到这个节点查看一下。查看之后，如果全是 us cpu 消耗，那我觉得这个基准测试就可以结束了。因为对于一个应用来说，us cpu 高本来就是非常合理的情况。

之前一个做第三方测试的人跑过来跟我说，甲方爸爸不喜欢看到 CPU 使用率太高，让他想尽一切办法把 CPU 降下来。可是他没有什么招，所以就来问我该怎么办。

我问他测试的目标是什么。他回答客户并不关心 TPS 啥的，只说要把 CPU 降下来。我说这简单，你把压力降下来，CPU 不就降下来了吗？本来以为只是一句调侃的话，结果他真去做了，并且还被客户接受了！后来我反思了一下，因为自己错误引导了性能行业的发展方向。

从职业的角度来说，我们对一些不懂的客户，最好要有一个良好的沟通，用对方能听懂的语言来解释。不过，在不该让步的时候，我们也不能让步。这才是专业的价值，不能是客户要什么，我们就给什么。

现在我们来看一下这个节点的 top 数据：

复制代码

```
[root@k8s-worker-8 ~]# top
top - 02:32:26 up 1 day, 13:56,  3 users,  load average: 26.46, 22.37, 14.54
Tasks: 289 total,   1 running, 288 sleeping,   0 stopped,   0 zombie
%Cpu0  : 73.9 us,  9.4 sy,  0.0 ni,  3.5 id,  0.0 wa,  0.0 hi, 12.5 si,  0.7 s
%Cpu1  : 69.8 us, 12.5 sy,  0.0 ni,  4.3 id,  0.0 wa,  0.0 hi, 12.8 si,  0.7 s
```

```
%Cpu2  : 71.5 us, 12.7 sy,  0.0 ni,  4.2 id,  0.0 wa,  0.0 hi, 10.9 si,  0.7 s
%Cpu3  : 70.3 us, 11.5 sy,  0.0 ni,  6.1 id,  0.0 wa,  0.0 hi, 11.5 si,  0.7 s
KiB Mem : 16266296 total,  3803848 free,  6779796 used,  5682652 buff/cache
KiB Swap:        0 total,        0 free,        0 used.  9072592 avail Mem


  PID USER      PR  NI    VIRT    RES    SHR S  %CPU %MEM     TIME+ COMMAND
30890 root      20   0 7791868 549328  15732 S 276.1  3.4  23:17.90 java -Dapp
18934 root      20   0 3716376   1.6g  18904 S  43.9 10.3 899:31.21 java -XX:+
 1059 root      20   0 2576944 109856  38508 S  11.1  0.7 264:59.42 /usr/bin/k
 1069 root      20   0 1260592 117572  29736 S  10.8  0.7 213:48.18 /usr/bin/d
15018 root      20   0 5943032   1.3g  16496 S   6.9  8.6 144:47.90 /usr/lib/j
 4723 root      20   0 1484184  43396  17700 S   5.9  0.3  89:53.45 calico-nod
```

在这个例子中，我们看到 si cpu（软中断消耗的 CPU）有 10% 多，其实这只是一个瞬间值，在不断跳跃的数据中，有很多次数据都比这个值大，说明 si 消耗的 CPU 有点高了。对于这种数据，我们就要关心一下了。

定向监控分析

我们进一步来看软中断的变化，既然软中断消耗的 CPU 高，那必然是要看一下软中断的计数器了：

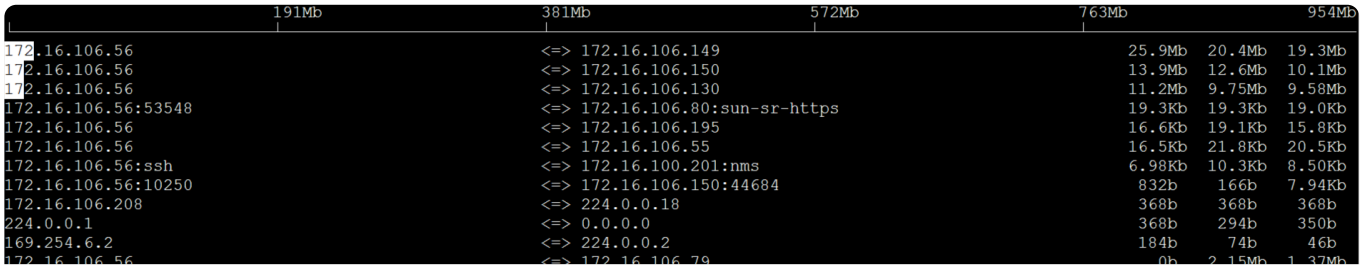
```
Every 1.0s: cat /proc/softirqs

                    CPU0       CPU1       CPU2       CPU3
          HI:          3          2          3          3
       TIMER:   59024803   59907994   59565591   58848924
      NET_TX:         30         25         42         52
      NET_RX:   62900054   61705553   60455464   60016457
       BLOCK:      85856      90290      95399     107824
BLOCK_IOPOLL:          0          0          0          0
     TASKLET:      14549      13069      12646      13469
       SCHED:   24649649   22260618   20334437   19161860
     HRTIMER:          0          0          0          0
         RCU:   28926774   29097811   28914249   28667240
```

上图是一张瞬间的截图，而在实际的观察过程中，我们是要多看一会儿时间的。请你注意图中这些有白底的数字，在观察中，这些数值增加的越大说明中断越高。而我在观察的过程中，看到的是 NET_RX 变化的最大。

现在，从 si cpu 高到 NET_RX 中断多的逻辑基本上清楚了：因为 NET_RX 都是网络的接收，所以 NET_RX 会不断往上跳。

不过，请你注意，这个中断即使是正常的，也需要不断增加。我们要判断它合理不合理，一定要结合 si cpu 一起来看。并且在网络中断中，不止是软中断，硬中断也会不断增加。

从上图来看，网络中断已经均衡了，没有单队列网卡的问题。我们再看一下网络带宽。



总共用了 50Mb 的带宽，中断就已经达到 10%，也就是说带宽没有完全用满，可是中断已经不低了，这说明我们的数据包中还是小包居多。

于是我们做如下调整。调整的方向 就是增加队列长度和缓冲区大小，让应用可以接收更多的数据包。

复制代码

```
-- 增加网络的队列长度
net.core.netdev_max_backlog = 10000  (原值: 1000)
- 增加tomcat的队列长度为10000(原值1000)
server:
  port: 8083
  tomcat
    accept-count: 10000
-- 改变设备一次可接收的数据包数量
net.core.dev_weight = 128  (原值64)
-- 控制socket 读取位于等待设备队列中数据包的微秒数
net.core.busy_poll = 100  (原值0)
-- 控制 socket 使用的接收缓冲区的默认大小
```

```
net.core.rmem_default = 2129920   (原值：212992)
net.core.rmem_max = 2129920   (原值：212992)
-- 繁忙轮询
 net.core.busy_poll = 100
 这个参数是用来控制了socket 读取位于等待设备队列中数据包的微秒数
```

一顿操作猛如虎之后，原本满怀希望，然而再次查了 TPS 曲线之后，发现并没有什么卵用，让我们把一首《凉凉》唱出来。

我仔细想了一遍发送和接收数据的逻辑。既然上层应用会导致 us cpu 高，而 si cpu 高是因为网卡中断多引起的，那我们还是要从网络层下手。所以，我做了网络带宽能达到多高的验证。我先列一下当前的硬件配置。

机器	类型	配置
A1	物理机	16C32G
B1	物理机	32C64G
A2	物理机A1+KVM	4C16G
B2	物理机B1+KVM	6C16G

我们通过 iperf3 直接测试网络，试验内容如下：

试验	方向	包大小	带宽	发送方si cpu(平均)	接收方si cpu(平均)
1	A1 – B1	4K	941Mb	0.6%	0%
2	A2 – B2	4K	927Mb	10%	2%
3	A3 – B3	4K	400Mb	20%	5%

从上面的数据可以看到，在不同的层面进行纯网络测试，si 是有很大区别的。当网络流量走了 KVM+Kubernetes+Docker 的结构之后，网络损失居然这么高，si cpu 也上升了很多。

这也解释了为什么现在很多企业放弃虚拟化，直接用物理机来跑 Kubernetes 了。

由于当前 K8s 用的是 Calico 插件中的 IPIP 模式，考虑到 BGP 模式的效率会高一些，我们把 IPIP 模式改为 BGP。这一步也是为了降低网络接收产生的软中断。

那 IPIP 和 BGP 到底有什么区别呢？对于 IPIP 来说，它套了两次 IP 包，相当于用了一个 IP 层后，还要用另一个 IP 层做网桥。在通常情况下，IP 是基于 MAC 的，不需要网桥；而 BGP 是通过维护路由表来实现对端可达的，不需要多套一层。但是 BGP 不是路由协议，而是矢量性协议。关于 IPIP 和 BGP 更多原理上的区别，如果你不清楚，我建议你自学一下相关的网络基础知识。

我们把 IPIP 修改为 BGP 模式之后，先测试下纯网络的区别，做这一步是为了看到在没有应用压力流量时，网络本身的传输效率如何：

路由 bgp

```
iperf Done.
bash-4.2# iperf3 -c svc-mall-member -p 12345
Connecting to host svc-mall-member, port 12345
[  4] local 10.100.230.2 port 48698 connected to 10.96.68.9 port 12345
[ ID] Interval           Transfer     Bandwidth       Retr  Cwnd
[  4]   0.00-1.00   sec   108 MBytes   907 Mbits/sec   80   1.46 MBytes
[  4]   1.00-2.00   sec   105 MBytes   881 Mbits/sec    0   1.51 MBytes
[  4]   2.00-3.00   sec   111 MBytes   933 Mbits/sec    0   1.56 MBytes
[  4]   3.00-4.00   sec   111 MBytes   933 Mbits/sec    0   1.61 MBytes
[  4]   4.00-5.00   sec   112 MBytes   943 Mbits/sec    0   1.66 MBytes
[  4]   5.00-6.00   sec   111 MBytes   934 Mbits/sec    0   1.70 MBytes
[  4]   6.00-7.00   sec   109 MBytes   912 Mbits/sec    0   1.75 MBytes
[  4]   7.00-8.00   sec   111 MBytes   933 Mbits/sec   46   1.79 MBytes
[  4]   8.00-9.00   sec   110 MBytes   923 Mbits/sec    0   1.83 MBytes
[  4]   9.00-10.00  sec   111 MBytes   933 Mbits/sec    0   1.88 MBytes
- - - - - - - - - - - - - - - - - - - - - - - - -
[ ID] Interval           Transfer     Bandwidth       Retr
[  4]   0.00-10.00  sec  1.07 GBytes   923 Mbits/sec  126             sender
[  4]   0.00-10.00  sec  1.07 GBytes   923 Mbits/sec                  receiver

iperf Done.
bash-4.2# iperf3 -c svc-mall-member -p 12345 -i 1 -t 10 -w 4K
Connecting to host svc-mall-member, port 12345
[  4] local 10.100.230.2 port 48726 connected to 10.96.68.9 port 12345
[ ID] Interval           Transfer     Bandwidth       Retr  Cwnd
[  4]   0.00-1.00   sec  6.97 MBytes  58.5 Mbits/sec    0   13.6 KBytes
[  4]   1.00-2.00   sec  5.20 MBytes  43.6 Mbits/sec    0   13.6 KBytes
[  4]   2.00-3.00   sec  6.63 MBytes  55.6 Mbits/sec    0   13.6 KBytes
[  4]   3.00-4.00   sec  6.46 MBytes  54.2 Mbits/sec    0   13.6 KBytes
[  4]   4.00-5.00   sec  5.48 MBytes  46.0 Mbits/sec    0   13.6 KBytes
[  4]   5.00-6.00   sec  5.68 MBytes  47.7 Mbits/sec    0   13.6 KBytes
[  4]   6.00-7.00   sec  6.43 MBytes  53.8 Mbits/sec    0   13.6 KBytes
[  4]   7.00-8.00   sec  7.01 MBytes  59.0 Mbits/sec    0   13.6 KBytes
[  4]   8.00-9.00   sec  6.38 MBytes  53.5 Mbits/sec    0   13.6 KBytes
[  4]   9.00-10.00  sec  6.22 MBytes  52.2 Mbits/sec    0   13.6 KBytes
- - - - - - - - - - - - - - - - - - - - - - - - -
[ ID] Interval           Transfer     Bandwidth       Retr
[  4]   0.00-10.00  sec  62.5 MBytes  52.4 Mbits/sec    0             sender
[  4]   0.00-10.00  sec  62.5 MBytes  52.4 Mbits/sec                  receiver

iperf Done.
bash-4.2# iperf3 -c svc-mall-member -p 12345 -i 1 -t 10 -w 100K
```

隧道 ipip

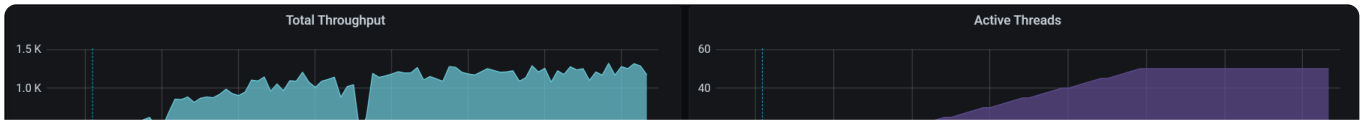
```
iperf Done.
bash-4.2# iperf3 -c svc-mall-member -p 12345
Connecting to host svc-mall-member, port 12345
[  4] local 10.100.230.2 port 43746 connected to 10.96.68.9 port 12345
[ ID] Interval           Transfer     Bandwidth       Retr  Cwnd
[  4]   0.00-1.00   sec  78.4 MBytes   657 Mbits/sec   11    317 KBytes
[  4]   1.00-2.01   sec  97.7 MBytes   816 Mbits/sec   68    469 KBytes
[  4]   2.01-3.00   sec  87.8 MBytes   736 Mbits/sec  137    567 KBytes
[  4]   3.00-4.00   sec  93.7 MBytes   789 Mbits/sec  387    476 KBytes
[  4]   4.00-5.00   sec  93.6 MBytes   786 Mbits/sec  172    579 KBytes
[  4]   5.00-6.00   sec   102 MBytes   852 Mbits/sec   87    678 KBytes
[  4]   6.00-7.00   sec  97.0 MBytes   814 Mbits/sec  102    751 KBytes
[  4]   7.00-8.00   sec  84.1 MBytes   705 Mbits/sec  216    725 KBytes
[  4]   8.00-9.00   sec   104 MBytes   876 Mbits/sec  133    876 KBytes
[  4]   9.00-10.00  sec  98.5 MBytes   826 Mbits/sec  273    923 KBytes
- - - - - - - - - - - - - - - - - - - - - - - - -
[ ID] Interval           Transfer     Bandwidth       Retr
[  4]   0.00-10.00  sec   937 MBytes   786 Mbits/sec  1586             sender
[  4]   0.00-10.00  sec   935 MBytes   785 Mbits/sec                   receiver

iperf Done.
bash-4.2# iperf3 -c svc-mall-member -p 12345 -i 1 -t 10 -w 4K
Connecting to host svc-mall-member, port 12345
[  4] local 10.100.230.2 port 43920 connected to 10.96.68.9 port 12345
[ ID] Interval           Transfer     Bandwidth       Retr  Cwnd
[  4]   0.00-1.00   sec  3.54 MBytes  29.7 Mbits/sec    0   13.6 KBytes
[  4]   1.00-2.00   sec  4.55 MBytes  38.1 Mbits/sec    0   13.6 KBytes
[  4]   2.00-3.00   sec  3.75 MBytes  31.5 Mbits/sec    1   13.6 KBytes
[  4]   3.00-4.00   sec  4.36 MBytes  36.6 Mbits/sec    0   13.6 KBytes
[  4]   4.00-5.00   sec  4.93 MBytes  41.4 Mbits/sec    0   13.6 KBytes
[  4]   5.00-6.00   sec  4.88 MBytes  40.9 Mbits/sec    0   13.6 KBytes
[  4]   6.00-7.00   sec  4.25 MBytes  35.6 Mbits/sec    0   13.6 KBytes
[  4]   7.00-8.00   sec  4.16 MBytes  34.9 Mbits/sec    1   13.6 KBytes
[  4]   8.00-9.00   sec  3.72 MBytes  31.2 Mbits/sec    0   13.6 KBytes
[  4]   9.00-10.00  sec  4.47 MBytes  37.4 Mbits/sec    0   13.6 KBytes
- - - - - - - - - - - - - - - - - - - - - - - - -
[ ID] Interval           Transfer     Bandwidth       Retr
[  4]   0.00-10.00  sec  42.6 MBytes  35.7 Mbits/sec    2             sender
[  4]   0.00-10.00  sec  42.6 MBytes  35.7 Mbits/sec                  receiver

iperf Done.
bash-4.2# iperf3 -c svc-mall-member -p 12345 -i 1 -t 10 -w 100K
```

根据上面的测试结果，将带宽在不同的网络模式和包大小时的具体数值整理如下：

网络模式	包大小	带宽

可以看到，BGP 的网络能力确实要强一些，差别好大呀。

我们再接着回去测试下接口，结果如下：



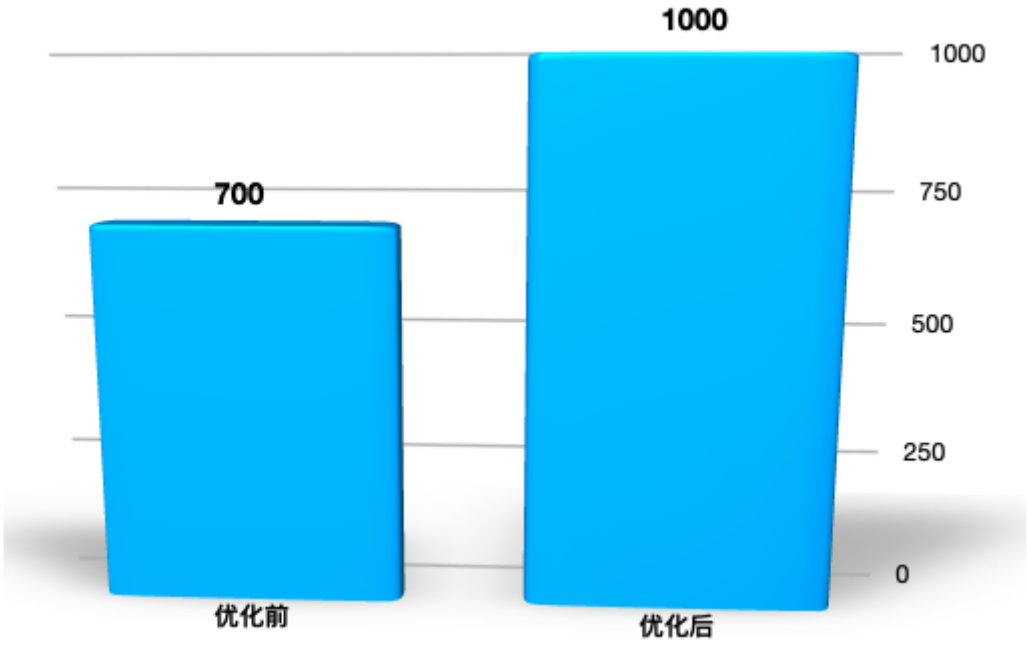
再看软中断，看一下 BGP 模式下的软中断有没有降低：

复制代码

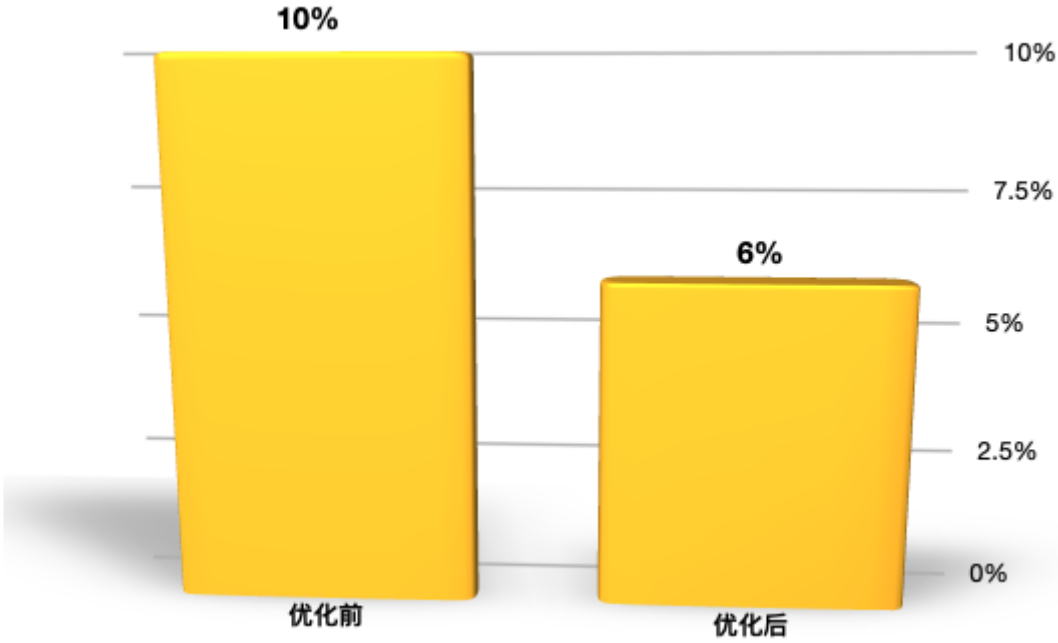
```
top - 22:34:09 up 3 days, 55 min,  2 users,  load average: 10.62, 6.18, 2.76
Tasks: 270 total,   2 running, 268 sleeping,   0 stopped,   0 zombie
%Cpu0  : 51.6 us, 11.5 sy,  0.0 ni, 30.0 id,  0.0 wa,  0.0 hi,  6.6 si,  0.3 s
%Cpu1  : 54.4 us,  9.4 sy,  0.0 ni, 28.2 id,  0.0 wa,  0.0 hi,  7.7 si,  0.3 s
%Cpu2  : 55.9 us, 11.4 sy,  0.0 ni, 26.9 id,  0.0 wa,  0.0 hi,  5.9 si,  0.0 s
%Cpu3  : 49.0 us, 12.4 sy,  0.0 ni, 32.8 id,  0.3 wa,  0.0 hi,  5.2 si,  0.3 s
KiB Mem : 16266296 total,  7186564 free,  4655012 used,  4424720 buff/cache
KiB Swap:        0 total,        0 free,        0 used. 11163216 avail Mem
```

优化效果

通过上面的调整结果，我们看到了软中断确实降低了不少，但是我们还是希望这样的优化体现到 TPS 上来，所以我们看一下优化之后 TPS 的效果。



si cpu 有降低：



总结

当我们看到一个接口已经满足了业务要求时，从成本上来说，我们不应该花时间再去收拾它。但是，**从技术上来说，我们对每一个接口的性能结果，都要达到“知道最终瓶颈在哪里”的程度**。这样才方便我们在后续的工作中继续优化。

在这节课的例子中，我们从 si cpu 开始分析，经过软中断查找和纯网络测试，定位到了 Kubernetes 的网络模式，进而我们选择了更加合理的网络模式。整个过程穿过了很长的链路，而这个思维也是在我在宣讲中一贯提到的“**证据链**”。

最后，我还是要强调一遍，**性能分析一定要有证据链，没有证据链的性能分析就是耍流氓。**我们要做正派的老司机。

课后作业

我给你留两道题，请你思考一下：

1. 为什么看到 NET_RX 中断高的时候，我们会想到去测试一下纯网络带宽？
2. 你能总结一下，这节课案例的证据链吗？

记得在留言区和我讨论、交流你的想法，每一次思考都会让你更进一步。

如果你读完这篇文章有所收获，也欢迎你分享给你的朋友，共同学习进步。我们下一讲再见！

提建议

© 版权归极客邦科技所有，未经许可不得传播售卖。 页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 13 | 用户登录：怎么判断线程中的Block原因？

下一篇 15 | 查询商品：资源不足有哪些性能表现？

精选留言 (2)

写留言

道长

2021-04-23

先通过top，查看到si高，再通过cat /proc/soft...(查看导致软中断)，查询这条的时候想看动态的变化情况，在命令前加watch cat...，查看到是RX(网络接收导致)，再查到中断消耗的带宽大小，并不大，进而想到是是由于网络小包导致，后面有点没太理解为什么想到了网络模式了问题。

展开

作者回复: 因为在这样的结构中。网络消耗不多，中断又高，所以，我们要查一下网络中断为什么那么高。对于底层网络的传输来说，中断是为了让数据可以正常让上层应用接收而用来一层层往上通知用的。
我们用的是KVM上面部署了K8s，k8s中的网络插件是有不同的网络模式的，所以这里只要想清楚这个层级关系，就明白了为什么要看网络模式了。对于k8s的优化，网络是非常重要的一块内容，在学习k8s的时候，你就会知道k8s网络模式有不同的实现方式，而这些实现方式的区别会导致中断数量有不同。

xhkk
2021-04-21

总共用了 50Mb 的带宽，中断就已经达到 10%，也就是说带宽没有完全用满，可是中断已经不低了，这说明我们的数据包中还是小包居多。

这段话的内容是从该段话上面的图片得到的，高老师，请问图片中哪里显示了是50Mb带宽？中断达到10%又是从那个数字判断的？...

展开

作者回复: 50M的带宽在上面图中的最右下角。10%在上面的top命令中的si列。
命令是iftop。